3

Classes, Records, Structs, and Tuples

WHAT'S IN THIS CHAPTER?

- Pass by value and by reference
- Classes and members
- Records
- Structs
- Enum types
- ref, in, and out keywords
- Tuples
- Deconstruction
- Pattern Matching
- Partial Types

CODE DOWNLOADS FOR THIS CHAPTER

The source code for this chapter is available on the book page at `www.wiley.com`. Click the Downloads link. The code can also be found at `https://github.com/ProfessionalCSharp/ProfessionalCSharp2021` in the directory `1_CS/Types`.

The code for this chapter is divided into the following major examples:

- TypesSample
- ClassesSample
- MathSample
- MethodSample

- ExtensionMethods
- RecordsSample
- StructsSample
- EnumSample
- RefInOutSample
- TuplesSample
- PatternMatchingSample

All the projects have nullable reference types enabled.

CREATING AND USING TYPES

So far, you've been introduced to some of the building blocks of the C# language, including variables, data types, and program flow statements, and you have seen a few short but complete programs that contain little more than top-level statements and a few methods. What you haven't seen yet is how to put all these elements together to form a longer program. The key to this lies in working with the types of .NET—classes, records, structs, and tuples, which are the subject of this chapter.

> **NOTE** *This chapter introduces the basic syntax associated with types. However, I assume that you are already familiar with the underlying principles of using classes—for example, that you know what a constructor or a property is. This chapter is largely confined to applying those principles in C# code.*

PASS BY VALUE OR BY REFERENCE

The types available with .NET can be categorized as pass by reference or pass by value.

Pass by value means that if you assign a variable to another variable, the value is copied. If you change the new value, the original value does not change. The content of the variable is copied on assignment. With the following code sample, a struct is created that contains a public field `A`. `x1` and `x2` are variables of this type. After creating `x1`, `x2` is assigned to `x1`. Because struct is a value type, the data from `x2` is copied to `x1`. Changing the value of the public field with `x2` doesn't influence `x1` at all. The `x1` variable still lists the original value; the value was copied (code file `TypesSample/Program.cs`):

```
AStruct x1 = new() { A = 1 };
AStruct x2 = x1;
x2.A = 2;
Console.WriteLine($"original didn't change with a struct: {x1.A}");

//...

public struct AStruct
{
  public int A;
}
```

> **NOTE** *Usually, you should not create public fields; instead, you should use other members such as properties. To give you an easy view of major differences with the .NET types, public fields are used.*

This behavior is very different with classes. If you change the public member of `A` within the `y2` variable, using the reference `y1`, the new value assigned from `y2` can be read. *Pass by reference* means that the variables `y1` and `y2` after assignment reference the same object (code file `TypesSample/Program.cs`):

```
AClass y1 = new() { A = 1 };
AClass y2 = y1;
y2.A = 2;
Console.WriteLine($"original changed with a class: {y1.A}");

//...

public class AClass
{
  public int A;
}
```

Another difference between the types that's worth mentioning is where the data is stored. With a reference type like the class, the memory where the data is stored is the managed heap. The variable itself is on the stack and references the content on the heap. A value type like the struct is usually stored on the stack. This is important with regard to garbage collection. The garbage collector needs to clean up objects in the heap if they are no longer used. Memory on the stack is automatically released at the end of the method, when the variable is outside of its scope.

> **NOTE** *The value of a struct typically is stored on the stack. However, with boxing—that is, when a struct is passed as an object or object methods are invoked with the struct—the data of the struct is moved to the heap. Boxing moves a struct to the heap; unboxing moves it back. C# also has a type where the data can never be stored on the heap; this is the ref struct. With a ref struct, you get a compilation error if operations are used that would move the data to the heap. This is discussed in Chapter 13, "Managed and Unmanaged Memory."*

Let's take a look at the record type that's new with C# 9. Using the `record` keyword, a record is created. Similar to our previous example when the class keyword was used to create a reference type, with the record keyword a reference type is created as well. A C# 9 record is a class. This C# keyword is just "syntax sugar": the compiler creates a class behind the scenes. There's no functionality needed from the runtime; you could create the same generated code without using this keyword, you just would need a lot more code lines (code file `TypesSample/Program.cs`):

```
ARecord z1 = new() { A = 1 };
ARecord z2 = z1;
z2.A = 2;
Console.WriteLine($"original changed with a record: {z1.A}");

//...
```

```
public record ARecord
{
  public int A;
}
```

> **NOTE** *A record supports value semantics like a struct, but it's implemented as a class. This comes from the fact that a record offers easy creation of immutable types and has members that cannot be changed after initialization. Read more about records later in this chapter.*

What about tuples? With tuples, you combine multiple types into one type without needing to create a class, struct, or record. How does this type behave?

In the following code snippet, `t1` is a tuple that combines a number and a string. The tuple `t1` is then assigned to the variable `t2`. If you change the value of `t2`, `t1` is not changed. The reason is that behind the scenes, using the C# syntax for tuples, the compiler makes use of the `ValueTuple` type—which is a struct—and copies values (code file `TypesSample/Program.cs`):

```
var t1 = (Number: 1, String: "a");
var t2 = t1;
t2.Number = 2;
t2.String = "b";
Console.WriteLine($"original didn't change with a tuple: {t1.Number} {t1.String}");
```

> **NOTE** *.NET offers the* `Tuple<T>` *type as well as the* `ValueTuple<T>` *type.* `Tuple<T>` *is the older one that is implemented as a class. With the built-in C# syntax for tuples, the* `ValueTuple` *is used. This* `ValueTuple` *contains public fields for all the members of the tuple. The old* `Tuple<T>` *type contains public read-only properties where the values cannot be changed. Today, there's no need to use the* `Tuple<T>` *type in your applications because there's better built-in support for* `ValueTuple<T>`.

Now that you've been introduced to the main differences between classes, structs, records, and tuples, let's dive deeper into the classes, including the members of classes. Most of the members of classes you learn about also apply to records and structs. I discuss the differences between records and structs after I introduce the members of classes.

CLASSES

A class contains members, which can be static or instance. A *static member* belongs to the class; an *instance member* belongs to the object. With static fields, the value of the field is the same for every object. With instance fields, every object can have a different value. Static members have the `static` modifier attached.

The kinds of members are explained in the following table:

MEMBER	DESCRIPTION
Fields	A field is a data member of a class. It is a variable of a type that is a member of a class.
Constants	Constants are associated with the class (although they do not have the `static` modifier). The compiler replaces constants everywhere they are used with the real value.

MEMBER	DESCRIPTION
Methods	Methods are functions associated with a particular class.
Properties	Properties are sets of functions that can be accessed from the client in a similar way to the public fields of the class. C# provides a specific syntax for implementing read and write properties on your classes, so you don't have to use method names that are prefixed with the words `Get` or `Set`. Because there's a dedicated syntax for properties that is distinct from that for normal functions, the illusion of objects as actual things is strengthened for client code.
Constructors	Constructors are special functions that are called automatically when an object is instantiated. They must have the same name as the class to which they belong and cannot have a return type. Constructors are useful for initialization.
Indexers	Indexers allow your object to be accessed the same way as arrays. Indexers are explained in Chapter 5, "Operators and Casts."
Operators	Operators, at their simplest, are actions such as + or –. When you add two integers, you are, strictly speaking, using the + operator for integers. C# also allows you to specify how existing operators will work with your own classes (operator overloading). Chapter 5 looks at operators in detail.
Events	Events are class members that allow an object to notify a subscriber whenever something noteworthy happens, such as a field or property of the class changing, or some form of user interaction occurring. The client can have code, known as an *event handler*, that reacts to the event. Chapter 7, "Delegates, Lambdas, and Events," looks at events in detail.
Destructors	The syntax of destructors or finalizers is similar to the syntax for constructors, but they are called when the CLR detects that an object is no longer needed. They have the same name as the class, preceded by a tilde (~). It is impossible to predict precisely when a finalizer will be called. Finalizers are discussed in Chapter 13.
Deconstructors	Deconstructors allow you to deconstruct the object into a tuple or different variables. Deconstruction is explained later in the section "Deconstruction."
Types	Classes can contain inner classes. This is interesting if the inner type is used only in conjunction with the outer type.

Let's get into the details of class members.

Fields

Fields are any variables associated with the class. In the class `Person`, the fields `_firstName` and `_lastName` of type `string` are defined. It's a good practice to declare fields with the `private` access modifier, which only allows accessing fields from within the class (code file `ClassesSample/Person.cs`):

```
public class Person
{
```

```
  //...
  private string _firstName;
  private string _lastName;
  //...
}
```

> **NOTE** *Members declared with the* `private` *access modifier only allow members of the class to invoke this member. To allow access from everywhere, use the* `public` *access modifier. Besides these two modifiers, C# also defines* `internal` *and* `protected` *to be used with access modifiers. All the different access modifiers are explained in detail in Chapter 4, "Object-Oriented Programming in C#."*

In the class `PeopleFactory`, the field `s_peopleCount` is of type `int` and has the `static` modifier applied. With the `static` modifier, the field is used with all instances of the class. Instance fields (without the `static` modifier) have different values for every instance of the class. Because this class only has static members, the class itself can have the `static` modifier applied. The compiler than makes sure that instance members are not added (code file `ClassesSample/PeopleFactory.cs`):

```
public static class PeopleFactory
{
  //...
  private static int s_peopleCount;
  //...
}
```

Readonly Fields

To guarantee that fields of an object cannot be changed, fields can be declared with the `readonly` modifier. Fields with the `readonly` modifier can be assigned only values from constructors. This is different from the `const` modifier shown in Chapter 2, "Core C#." With the `const` modifier, the compiler replaces the variable by its value everywhere it is used. The compiler already knows the value of the constant. Read-only fields are assigned during runtime from a constructor. The following `Person` class specifies a constructor where values for both `firstName` and `lastName` need to be passed.

Contrary to const fields, read-only fields can be instance members. With the following code snippet, the `_firstName` and `_lastName` fields are changed to add the `readonly` modifier. The compiler complains with errors if this field is changed after initializing it in the constructor (code file `ClassesSample/Person.cs`):

```
public class Person
{
  //...
  public Person(string firstName, string lastName)
  {
    _firstName = firstName;
    _lastName = lastName;
  }

  private readonly string _firstName;
  private readonly string _lastName;
  //...
}
```

Properties

Instead of having a method pair to set and get the values of a field, C# defines the syntax of a property. From outside of the class, a property looks like a field with typically used uppercase names. Within the class, you can write a custom implementation to set not just fields and get the value of fields, but you can add some programming logic to validate the value before assigning it to a variable. You can also define a purely computed property without any variable that is accessed by the property.

The class `Person` as shown in the following code snippet defines a property with the name `Age` accessing the private field `_age`. With the `get` accessor, the value of the field is returned. With the `set` accessor, the variable `value`, which contains the value passed when setting the property, is automatically created. In the code snippet, the value variable is used to assign the value to the `_age` field (code file `ClassesSample/Person.cs`):

```
public class Person
{
  //...

  private int _age;
  public int Age
  {
    get => _age;
    set => _age = value;
  }
}
```

In case more than one statement is needed with the implementation of the property accessor, you can use curly brackets as shown in the following code snippet:

```
private int _age;
public int Age
{
  get
  {
    return _age;
  }
  set
  {
    _age = value;
  }
}
```

To use the property, you can access the property from an object instance. Setting a value to the property invokes the `set` accessor. Reading the value invokes the `get` accessor:

```
person.Age = 4; // setting a property value with the set accessor
int age = person.Age; // accessing the property with the get accessor
```

Auto-Implemented Properties

If there isn't going to be any logic in the property accessors `set` and `get`, then auto-implemented properties can be used. Auto-implemented properties implement the backing member variable automatically. The code for the earlier `Age` example would look like this:

```
public int Age { get; set; }
```

The declaration of a private field is not needed. The compiler creates this automatically. With auto-implemented properties, you cannot access the field directly because you don't know the name the compiler generates. If all

you need to do with a property is read and write a field, the syntax for the property using auto-implemented properties is shorter than using expression-bodied property accessors.

By using auto-implemented properties, validation of the property cannot be done at the property set. Therefore, with the `Age` property, you could not have checked to see whether an invalid age is set.

Auto-implemented properties can be initialized using a property initializer. The compiler moves this initialization to the created constructor, and the initialization is done before the constructor body.

```
public int Age { get; set; } = 42;
```

Access Modifiers for Properties

C# allows the `set` and `get` accessors to have differing access modifiers. This would allow a property to have a public `get` and a private or protected `set`. This can help control how or when a property can be set. In the following code example, notice that the `set` has a private access modifier, but the `get` does not. In this case, the `get` takes the access level of the property. One of the accessors must follow the access level of the property. A compile error is generated if the `get` accessor has the `protected` access level associated with it because that would make both accessors have a different access level from the property.

```
private string _name;
public string Name
{
  get => _name;
  private set => _name = value;
}
```

Different access levels can also be set with auto-implemented properties:

```
public int Age { get; private set; }
```

Readonly Properties

It is possible to create a read-only property by simply omitting the `set` accessor from the property definition. Thus, to make `FirstName` a read-only property, you can do this by just defining the `get` accessor:

```
private readonly string _firstName;
public string FirstName
{
  get => _firstName;
}
```

Declaring the field with the `readonly` modifier only allows initializing the value of the property in the constructor.

> **NOTE** *Similar to properties that only make use of a* `get` *accessor, you can also specify a property with just a* `set` *accessor. This is a write-only property. However, this is regarded as poor programming practice and could be confusing to the developers who access this property. It's recommended that you define methods instead of using write-only properties.*

Expression-Bodied Properties

With properties that only implement a `get` accessor, you can use a simplified syntax with the => token and assign an expression-bodied member. There's no need to write the `get` accessor to return a value. Behind the scenes, the compiler creates an implementation with a `get` accessor.

In the following code snippet, a `FirstName` property is defined that returns the field `_firstName` using an expression-bodied property. The `FullName` property combines the `_firstName` field and the value from the `LastName` property to return the full name (code file `ClassesSample/Person.cs`):

```
private readonly string _firstName;
public string FirstName => _firstName;
private readonly string _lastName;
public strign LastName => _lastName;
public string FullName => $"{FirstName} {LastName}";
```

Auto-Implemented Read-Only Properties

C# offers a simple syntax with auto-implemented properties to create read-only properties that access read-only fields. These properties can be initialized using property initializers:

```
public string Id { get; } = Guid.NewGuid().ToString();
```

Behind the scenes, the compiler creates a read-only field and a property with a `get` accessor to this field. The code from the initializer moves to the implementation of the constructor and is invoked before the constructor body is called.

Read-only properties can also explicitly be initialized from the constructor, as shown with this code snippet:

```
public class Book
{
  public Book(string title) => Title = title;

  public string Title { get; }
}
```

Init-Only Set Accessors

C# 9 allows you to define properties with `get` and `init` accessors by using the `init` keyword instead of the `set` keyword. This way the property value can be set only in the constructor or with an object initializer (code file `ClassesSample/Book.cs`):

```
public class Book
{
  public Book(string title)
  {
    Title = title;
  }

  public string Title { get; init; }
  public string? Publisher { get; init; }
}
```

C# 9 offers a new option with properties that should only be set with constructors and object initializers. A new `Book` object can now be created by invoking the constructor and using an object initializer to set the properties as shown in the following code snippet (code file `ClassesSample/Program.cs`):

```
Book theBook = new("Professional C#")
{
  Publisher = "Wrox Press"
};
```

You can use object initializers to initialize properties on creation of the object. The constructor defines the required parameters that the class needs for initialization. With the object initializer, you can assign all properties with a `set` and an `init` accessor. The object initializer can be used only when creating the object, not afterward.

Methods

With the C# terminology, there's a distinction between functions and methods. The term *function member* includes not only methods, but also other nondata members such as indexers, operators, constructors, destructors, and properties—all members that contain executable code.

Declaring Methods

In C#, the definition of a method consists of any method modifiers (such as the method's accessibility), followed by the type of the return value, followed by the name of the method, followed by a list of parameters enclosed in parentheses, followed by the body of the method enclosed in curly brackets.

Each parameter consists of the name of the type of the parameter and the name by which it can be referenced in the body of the method. Also, if the method returns a value, a return statement must be used with the return value to indicate each exit point, as shown in this example:

```
public bool IsSquare(Rectangle rect)
{
  return (rect.Height == rect.Width);
}
```

If the method doesn't return anything, specify a return type of `void` because you can't omit the return type altogether. If the method takes no parameters, you need to include an empty set of parentheses after the method name. With a void return, using a return statement in the implementation is optional—the method returns automatically when the closing curly brace is reached.

Expression-Bodied Methods

If the implementation of a method consists just of one statement, C# gives a simplified syntax to method definitions: *expression-bodied methods*. You don't need to write curly brackets and the `return` keyword with this syntax. The `=>` token is used to distinguish the declaration of the left side of this operator to the implementation that is on the right side.

The following example is the same method as before, `IsSquare`, implemented using the expression-bodied method syntax. The right side of the `=>` token defines the implementation of the method. Curly brackets and a return statement are not needed. What's returned is the result of the statement, and the result needs to be of the same type as the method declared on the left side, which is a `bool` in this code snippet:

```
public bool IsSquare(Rectangle rect) => rect.Height == rect.Width;
```

Invoking Methods

The following example illustrates the syntax for definition and instantiation of classes and for definition and invocation of methods. The class `Math` defines instance and static members (code file `MathSample/Math.cs`):

```
public class Math
{
  public int Value { get; set; }
  public int GetSquare() => Value * Value;
  public static int GetSquareOf(int x) => x * x;
}
```

The top-level statements in the `Program.cs` file uses the `Math` class, calls static methods, and instantiates an object to invoke instance members (code file `MathSample/Program.cs`):

```
using System;
```

```
// Call static members
int x = Math.GetSquareOf(5);
Console.WriteLine($"Square of 5 is {x}");

// Instantiate a Math object
Math math = new();

// Call instance members
math.Value = 30;
Console.WriteLine($"Value field of math variable contains {math.Value}");
Console.WriteLine($"Square of 30 is {math.GetSquare()}");
```

Running the `MathSample` example produces the following results:

```
Square of 5 is 25
Value field of math variable contains 30
Square of 30 is 900
```

As you can see from the code, the `Math` class contains a property that contains a number, as well as a method to find the square of this number. It also contains one static method to find the square of the number passed in as a parameter.

Method Overloading

C# supports method overloading—several versions of the method that have different signatures (that is, the same name but a different number of parameters and/or different parameter data types). To overload methods, simply declare the methods with the same name but different numbers of parameter types:

```
class ResultDisplayer
{
  public void DisplayResult(string result)
  {
    // implementation
  }

  public void DisplayResult(int result)
  {
    // implementation
  }
}
```

It's not just the parameter types that can differ; the number of parameters can differ too, as shown in the next example. One overloaded method can invoke another:

```
class MyClass
{
  public int DoSomething(int x) => DoSomething(x, 10);

  public int DoSomething(int x, int y)
  {
    // implementation
  }
}
```

> **NOTE** *With method overloading, it is not sufficient to differ overloads only by the return type. It's also not sufficient to differ them by parameter names. The number of parameters and/or types needs to differ.*

Named Arguments

When invoking methods, the variable name need not be added to the invocation. However, if you have a method signature like the following to move a rectangle:

```
public void MoveAndResize(int x, int y, int width, int height)
```

and you invoke it with the following code snippet, it's not clear from the invocation what numbers are used for what:

```
r.MoveAndResize(30, 40, 20, 40);
```

You can change the invocation to make it immediately clear what the numbers mean:

```
r.MoveAndResize(x: 30, y: 40, width: 20, height: 40);
```

Any method can be invoked using named arguments. You just need to write the name of the variable followed by a colon and the value passed. The compiler gets rid of the name and creates an invocation of the method just as if the variable name is not there—so there's no difference within the compiled code.

You can also change the order of variables this way, and the compiler rearranges it to the correct order. The real advantage to this is shown in the next section with optional arguments.

Optional Arguments

Parameters can also be optional. You must supply a default value for optional parameters, which must be the last ones defined:

```
public void TestMethod(int notOptionalNumber, int optionalNumber = 42)
{
  Console.WriteLine(optionalNumber + notOptionalNumber);
}
```

This method can now be invoked using one or two parameters. When you pass one parameter, the compiler changes the method call to pass `42` with the second parameter:

```
TestMethod(11);
TestMethod(11, 42);
```

> **NOTE** *Because the compiler changes methods with optional parameters to pass the default value, the default value should never change with newer versions of the library. This is a breaking change because the calling application can still have the previous value without recompilation.*

You can define multiple optional parameters, as shown here:

```
public void TestMethod(int n, int opt1 = 11, int opt2 = 22, int opt3 = 33)
{
  Console.WriteLine(n + opt1 + opt2 + opt3);
}
```

This way, the method can be called using one, two, three, or four parameters. The first line of the following code leaves the optional parameters with the values `11`, `22`, and `33`. The second line passes the first three parameters, and the last one has a value of `33`:

```
TestMethod(1);
TestMethod(1, 2, 3);
```

With multiple optional parameters, the feature of named arguments shines. When you use named arguments, you can pass any of the optional parameters. For example, this example passes just the last one:

```
TestMethod(1, opt3: 4);
```

> **WARNING** *Pay attention to versioning issues when using optional arguments. One issue is changing default values in newer versions; another issue is changing the number of arguments. It might be tempting to add another optional parameter because it is optional anyway. However, the compiler changes the calling code to fill in all the parameters, and that's the reason why earlier compiled callers fail if another parameter is added later.*

Variable Number of Arguments

When you use optional arguments, you can define a variable number of arguments. However, there's also a different syntax that allows passing a variable number of arguments—and this syntax doesn't have versioning issues.

When you declare the parameter of type array—the sample code uses an `int` array—and add the `params` keyword, the method can be invoked using any number of `int` parameters.

```
public void AnyNumberOfArguments(params int[] data)
{
  foreach (var x in data)
  {
    Console.WriteLine(x);
  }
}
```

> **NOTE** *Arrays are explained in detail in Chapter 6, "Arrays."*

Because the parameter of the method `AnyNumberOfArguments` is of type `int[]`, you can pass an `int` array, or because of the `params` keyword, you can pass zero or more `int` values:

```
AnyNumberOfArguments(1);
AnyNumberOfArguments(1, 3, 5, 7, 11, 13);
```

If arguments of different types should be passed to methods, you can use an `object` array:

```
public void AnyNumberOfArguments(params object[] data)
{
  // ...
```

Now it is possible to use any type for the parameters calling this method:

```
AnyNumberOfArguments("text", 42);
```

If the `params` keyword is used with multiple parameters that are defined with the method signature, `params` can be used only once, and it must be the last parameter:

```
Console.WriteLine(string format, params object[] arg);
```

> **NOTE** *In case you've overloaded methods, and one of these methods is using the* `params` *keyword, the compiler prefers fixed parameters rather than the* `params` *keyword. For example, if a method is declared with two* `int` *parameters (*`Foo(int, int)`*), and another method is using the* `params` *keyword (*`Foo(int[] params)`*), when invoking this method with two* `int` *arguments, the method* `Foo(int, int)` *wins because it has a better match.*

Now that you've looked at the many aspects of methods, let's get into constructors, which are a special kind of method.

Constructors

The syntax for declaring basic constructors is a method that has the same name as the containing class and that does not have any return type:

```
public class MyClass
{
  public MyClass()
  {
  }

  //...
}
```

It's not necessary to provide a constructor for your class. If you don't supply any constructor, the compiler generates a default behind the scenes. This constructor initializes all the member fields to the default values, which is `0` for numbers, `false` for `bool`, and `null` for reference types. When you're using nullable reference types and don't declare your reference types to allow `null`, you'll get a compiler warning if these fields are not initialized.

Constructors follow the same rules for overloading as other methods—that is, you can provide as many overloads to the constructor as you want, provided they are clearly different in signature:

```
public MyClass() // parameterless constructor
{
  // construction code
}

public MyClass(int number) // constructor overload with an int parameter
{
  // construction code
}
```

If you supply any constructors, the compiler does not automatically supply a default one. The default constructor is created only if other constructors are not defined.

Note that it is possible to define constructors as `private` or `protected` so that they are invisible to code in unrelated classes, too:

```
public class MyNumber
{
  private int _number;
  private MyNumber(int number) => _number = number;
  //...
}
```

An example in which this is useful is to create a singleton where an instance can be created only from a static factory method.

Expression Bodies with Constructors

If the implementation of a constructor just consists of a single expression, the constructor can be implemented with an expression-bodied implementation:

```
public class Singleton
{
  private static Singleton s_instance;
  private int _state;
  private Singleton(int state) => _state = state;

  public static Singleton Instance => s_instance ??= new Singleton(42);
}
```

You can also initialize multiple properties with a single expression. You can do this using the tuple syntax as shown in the following code snippet. With the `Book` constructor, two parameters are required. Putting these two variables in parentheses creates a tuple. This tuple is then deconstructed and put into the properties specified with the left side of the assignment operator. Behind the scenes, the compiler detects that tuples are not needed for the initialization and creates the same code whether you initialize the properties within curly brackets or with the tuple syntax shown:

```
public class Book
{
  public Book(string title, string publisher) =>
    (Title, Publisher) = (title, publisher);

  public string Title { get; }
  public string Publisher { get; }
}
```

Calling Constructors from Other Constructors

When you're creating multiple constructors in a class, you shouldn't duplicate the implementation. Instead, one constructor can invoke another one from a constructor initializer.

Both constructors initialize the same fields. It would clearly be tidier to place all the code in one location. C# has a special syntax known as a *constructor initializer* to enable this:

```
class Car
{
  private string _description;
  private uint _nWheels;
  public Car(string description, uint nWheels)
  {
    _description = description;
    _nWheels = nWheels;
  }
  public Car(string description) : this(description, 4)
  {
  }
}
```

In this context, the `this` keyword simply causes the constructor with the matching parameters to be called. Note that any constructor initializer is executed before the body of the constructor.

Static Constructors

Static members of a class can be used before any instance of this class is created (if any instance is created at all). To initialize static members, you can create a static constructor. The static constructor has the same name as the class (similar to an instance constructor), but the `static` modifier is applied. This constructor cannot have an access modifier applied because it isn't invoked from the code using the class. This constructor is automatically invoked before any other member of this class is called or any instance is created:

```
class MyClass
{
  static MyClass()
  {
    // initialization code
  }
  //...
}
```

The .NET runtime makes no guarantees about when a static constructor will be executed, so you should not place any code in it that relies on it being executed at a particular time (for example, when an assembly is loaded). Nor is it possible to predict in what order static constructors of different classes will execute. However, what is guaranteed is that the static constructor will run at most once, and it will be invoked before your code makes any reference to the class. In C#, the static constructor is usually executed immediately before the first call to any member of the class.

Local Functions

Methods with a `public` access modifier can be invoked from outside of the class. Methods with a `private` access modifier can be invoked from anywhere within the class (from other methods, property accessors, constructors, and so on). To restrict this further, a *local function* can be invoked only from within the method where the local function is declared. The local function has the scope of the method and cannot be invoked from somewhere else.

Within the method `IntroLocalFunctions`, the local function `Add` is defined. Parameters and return types are implemented in the same way as a normal method. Similarly to a normal method, a local function can be implemented by using curly brackets or with an expression-bodied implementation as shown in the following code. Since C# 8, the local function can have the `static` modifier associated if the implementation doesn't access instance members defined with the class or local variables of the method. With the `static` modifier, the compiler makes sure this does not happen and can optimize the generated code. The local function is invoked in the method itself; it cannot be invoked anywhere else in the class. Whether the local function is declared before or after its use is just a matter of taste (code file `MethodSample/LocalFunctionsSample.cs`):

```
public static void IntroLocalFunctions()
{
  static int Add(int x, int y) => x + y;

  int result = Add(3, 7);
  Console.WriteLine("called the local function with this result: {result}");
}
```

With the next code snippet, the local function `Add` is declared without the `static` modifier. In the implementation, this function not only uses the variables specified with the arguments of the function but also variable `z`, which is specified in the outer scope of the local function, within the scope of the method. When accessing the variable outside of its scope (known as *closure*), the compiler creates a class where the data used within this function is passed in a constructor. Here, the local function needs to be declared after the variables used within the local function. That's why the local function is put at the end of the method `LocalFunctionWithClosure`:

```
public static void LocalFunctionWithClosure()
{
  int z = 3;
```

```
    int result = Add(1, 2);
    Console.WriteLine("called the local function with this result: {result}");

    int Add(int x, int y) => x + y + z;
}
```

> **NOTE** *Local functions can help with error handling when using deferred execution with the* `yield` *statement. This is shown in Chapter 9, "Language Integrated Query." With C# 9, local functions can have the* `extern` *modifier. This is shown for invoking native methods in Chapter 13.*

Generic Methods

If you need implementations of methods that support multiple types, you can implement generic methods. The method `Swap<T>` defines `T` as a generic type that is used for two arguments and a local variable temp (code file `MeethodSample/GenericMethods.cs`):

```
class GenericMethods
{
  public static void Swap<T>(ref T x, ref T y)
  {
    T temp;
    temp = x;
    x = y;
    y = temp;
  }
}
```

> **NOTE** *It's a convention to use* `T` *for the name of the generic type. In case you need multiple generic types, you can use* `T1`, `T2`, `T3`, *and so on. For specific generic types, you can also add a name—for example,* `TKey` *and* `TValue` *for generic types representing the type of the key and the type of the value.*

> **NOTE** *With generic methods, you can also invoke members of the generic type other than members of the object class if you define constraints and define that the generic type needs to implement an interface or derive from a base class. This is explained in Chapter 4 where generic types are covered.*

Extension Methods

With extension methods, you can create methods that extend other types.

The following code snippet defines the method `GetWordCount` that is used to extend the `string` type. An extension method is not defined by the name of the class but instead by using the `this` modifier with the parameter.

`GetWordCount` extends the string type because the parameter with the `this` modifier (which needs to be the first parameter) is of type string. Extension methods need to be static and declared in a static class (code file `ExtensionMethods/StringExtensions.cs`):

```
public static class StringExtensions
{
  public static int GetWordCount(this string s) => s.Split().Length;
}
```

To use this extension method, the namespace of the extension class needs to be imported; then the method can be called in the same way as an instance method (code file `ExtensionMethods/Program.cs`):

```
string fox = "the quick brown fox jumped over the lazy dogs";
int wordCount = fox.GetWordCount();
Console.WriteLine($"{wordCount} words");
Console.ReadLine();
```

It might look like extension methods break object-oriented rules in regard to inheritance and encapsulation because methods can be added to an existing type without inheriting from it and without changing the type. However, you can only access public members. Extension methods are really just "syntax sugar" because the compiler changes the invocation of the method to call a static method that's passing the instance as the parameter, as shown here:

```
int wordCount = StringExtensions.GetWordCount(fox);
```

Why would you create extension methods instead of calling static methods? The code can become a lot easier to read. Just check into the extension methods implemented for LINQ (see Chapter 9) or the extension methods used to configure configuration and logging providers (see Chapter 15, "Dependency Injection and Configuration").

Anonymous Types

Chapter 2 discusses the `var` keyword in reference to implicitly typed variables. When used with the `new` keyword, you can create anonymous types. An *anonymous type* is simply a nameless class that inherits from `object`. The definition of the class is inferred from the initializer, just as with implicitly typed variables.

For example, if you need an object that contains a person's first, middle, and last name, the declaration would look like this:

```
var captain = new
{
  FirstName = "James",
  MiddleName = "Tiberius",
  LastName = "Kirk"
};
```

This would produce an object with `FirstName`, `MiddleName`, and `LastName` read-only properties. If you were to create another object that looked like this:

```
var doctor = new
{
  FirstName = "Leonard",
  MiddleName = string.Empty,
  LastName = "McCoy"
};
```

then the types of `captain` and `doctor` are the same. You could set `captain = doctor`, for example. This is possible only if all the properties match.

The names for the members of anonymous types can be inferred if the values that are being set come from another object. This way, the initializer can be abbreviated. If you already have a class that contains the properties `FirstName`, `MiddleName`, and `LastName` and you have an instance of that class with the instance name `person`, then the `captain` object could be initialized like this:

```
var captain = new
{
  person.FirstName,
  person.MiddleName,
  person.LastName
};
```

The property names from the `person` object are inferred in the new object named `captain`, so the object named `captain` has `FirstName`, `MiddleName`, and `LastName` properties.

The actual type name of anonymous types is unknown, which is where the name comes from. The compiler "makes up" a name for the type, but only the compiler is ever able to make use of it. Therefore, you can't and shouldn't plan on using any type reflection on the new objects because you will not get consistent results.

RECORDS

So far in this chapter, you've seen that records are reference types that support value semantics. This type allows reducing the code you need to write because the compiler automatically implements comparing records by value and gives some more features, which are explained in this section.

Immutable Types

A main use case for records is to create immutable types (although you can also create mutable types with records). An immutable type just contains members where the state of the type cannot be changed. You can initialize such a type in a constructor or with an object initializer, but you can't change any values afterward.

Immutable types are useful with multithreading. When you're using multiple threads to access the immutable object, you don't need to worry with synchronization because the values cannot change.

An example of an immutable type is the `String` class. This class does not define any member that is allowed to change its content. Methods such as `ToUpper` (which changes the string to uppercase) always return a new string, but the original string passed to the constructor remains unchanged.

Nominal Records

Records can be created in two kinds: nominal and positional records. A nominal record looks like a class just using the `record` keyword instead of the `class` keyword, as shown with the type `Book1`. Here, init-only set accessors are used to forbid state changes after an instance has been created (code file `RecordsSample/Program.cs`):

```
public record Book1
{
    public string Title { get; init; } = string.Empty;
    public string Publisher { get; init; } = string.Empty;
}
```

You can add constructors and all the other members you learned about in this chapter. The compiler just creates a class with the record syntax. What's different from classes is that the compiler creates some more functionality inside this class. The compiler overrides the `GetHashCode` and `ToString` methods of the base class `object`, creates methods and operator overloads to compare different values for equality, and creates methods to clone existing objects and create new ones where object initializers can be used to change some property values.

> **NOTE** *See Chapter 5 for information about operator overloads.*

Positional Records

The second way to implement a record is to use the positional record syntax. With this syntax, parentheses are used after the name of the record to specify the members. This syntax has the name *primary constructor*. The compiler creates a class from this code as well, with init-only set accessors for the types used with the primary constructor and a constructor with the same parameters to initialize the properties (code file `RecordsSample/Program.cs`):

```
public record Book2(string Title, string Publisher);
```

You can use curly brackets to add what you need to the already existing implementation—for example, by adding overloaded constructors, methods, or any other members you've seen earlier in this chapter:

```
public record Book2(string Title, string Publisher)
{
  // add your members, overloads
}
```

As the compiler creates a constructor with parameters, you can instantiate an object as you're used to—by passing the values to the constructor (code file `RecordsSample/Program.cs`):

```
Book2 b2 = new("Professional C#", "Wrox Press");
Console.WriteLine(b2);
```

Because the compiler creates a `ToString` method that is implicitly invoked by passing the variable to the `WriteLine` method, this is what's shown on the console: the name of the class followed by the property names with their values in curly brackets:

```
Book2 { Title = Professional C#, Publisher = Wrox Press }
```

With positional records, the compiler creates the same members as with nominal records and adds methods for deconstruction. Deconstruction is explained later in this chapter in the section "Deconstruction."

Equality Comparison with Records

With classes, the default implementation for equality is to compare the reference. Creating two new objects of the same type that are initialized to the same values are different because they reference different objects in the heap. This is different with records. With the equality implementation of records, two records are equal if their property values are the same.

In the following code snippet, two records that contain the same values are created. The `object.ReferenceEquals` method returns `false`, because these are two different references. Using the equal operator `==` returns `true` because this operator is implemented with the record type (code file `RecordsSample/Program.cs`):

```
Book1 book1a = new() { Title = "Professional C#", Publisher = "Wrox Press" };
Book1 book1b = new() { Title = "Professional C#", Publisher = "Wrox Press" };
if (!object.ReferenceEquals(book1a, book1b))
  Console.WriteLine("Two different references for equal records");

if (book1a == book1b)
  Console.WriteLine("Both records have the same values");
```

The record type implements the `IEquality` interface with the `Equals` method, as well as the equality `==` and the inequality `!=` operators.

With Expressions

Records make it easy to create immutable types, but there's a new feature with records for easily creating new record instances. The .NET Compiler Platform (also known by the name Roslyn) is built with immutable objects and many `With` methods to create new objects from existing ones. With the C# 9 enhancement, the `with` expressions, there's a lot of simplification that can be used by the Roslyn team. The code created with the record syntax includes a copy constructor and a Clone method with a hidden name where all the values of the existing object are copied to a new instance that's returned from this method. The `with` expression now makes use of this Clone method, and with the init-only set accessors, you can use object initialization to set the values that should be different.

```
var aNewBook = book1a with { Title = "Professional C# and .NET - 2024" };
```

> **NOTE** *See Chapter 4 for inheritance not only with classes but also with records.*

STRUCTS

So far, you have seen how classes and records offer a great way to encapsulate objects in your program. You have also seen how they are stored on the heap in a way that gives you much more flexibility in data lifetime but with a slight cost in performance. Objects stored in the heap require work from the garbage collector to remove the memory of the objects that are no longer needed. To reduce the work needed by the garbage collector, you can use the stack for smaller objects.

Chapter 2 discusses predefined value types such as `int` and `double`, which are represented as a struct type. You can create such structs on your own.

Just by using the `struct` keyword instead of the `class` keyword, the type is by default stored in the stack instead of the heap.

The following code snippet defines a struct called `Dimensions`, which simply stores the length and width of an item. Suppose you're writing a furniture-arranging program that enables users to experiment with rearranging their furniture on the computer, and you want to store the dimensions of each item of furniture. All you have is two numbers, which you'll find convenient to treat as a pair rather than individually. There is no need for a lot of methods, or for you to be able to inherit from the class, and you certainly don't want to have the .NET runtime go to the trouble of bringing in the heap, with all the performance implications, just to store two `doubles` (code file `StructsSample/Dimensions.cs`):

```
public readonly struct Dimensions
{
  public Dimensions(double length, double width)
  {
    Length = length;
    Width = width;
  }

  public double Length { get; }
  public double Width { get; }
  //...
}
```

> **NOTE** *If the members of a struct don't change any state (other than the constructors), the struct can be declared with the* `readonly` *modifier. The compiler will make sure you don't add any members that change state.*

Defining members for structs is done in the same way as defining them for classes and records. You've already seen a constructor with the `Dimensions` struct. The following code demonstrates adding the property `Diagonal` invoking the `Sqrt` method of the `Math` class (code file `StructsSample/Dimensions.cs`):

```
public struct Dimensions
{
  //...
  public double Diagonal => Math.Sqrt(Length * Length + Width * Width);
}
```

Structs make use of the previously discussed pass by value semantics, where values are copied. This is not the only difference with classes and records:

- Structs do not support inheritance. You can implement interfaces with structs but not derive from another struct.
- Structs always have a default constructor. With a class, if you define a constructor, the default constructor no longer gets generated. The struct type is different than a class. A struct always has a default constructor, and you cannot create a custom parameterless constructor.
- With a struct, you can specify how the fields are laid out in memory. This is examined in Chapter 13.
- Structs are stored on the stack or inline (if they are part of another object that is stored on the heap). When a struct is used as an object (for example, passed to an object parameter or an object-based method is invoked), boxing occurs, and the value is copied to the heap as well.

ENUM TYPES

An enumeration is a value type that contains a list of named constants, such as the `Color` type shown here. The enumeration type is defined by using the `enum` keyword:

```
public enum Color
{
  Red,
  Green,
  Blue
}
```

You can declare variables of `enum` types, such as the variable `c1`, and assign a value from the enumeration by setting one of the named constants prefixed with the name of the `enum` type (code file `EnumSample/Program.cs`):

```
void ColorSamples()
{
  Color c1 = Color.Red;
  Console.WriteLine(c1);

  //...
}
```

When you run the program, the console output shows `Red`, which is the constant value of the enumeration.

By default, the type behind the enum type is an int. You can change the underlying type to other integral types (byte, short, int, long with signed and unsigned variants). The values of the named constants are incremental values starting with 0, but you can change them to other values (code file EnumSample/Color.cs):

```
public enum Color : short
{
  Red = 1,
  Green = 2,
  Blue = 3
}
```

You can change a number to an enumeration value and back using casts.

```
Color c2 = (Color)2;
short number = (short)c2;
```

You can also use an enum type to assign multiple options to a variable and not just one of the enum constants. To make exclusive enum values, the numbers assigned to the values should each set a single different bit.

The enum type DaysOfWeek defines different values for every day. Setting different bits can be done easily using hexadecimal values that are assigned using the 0x prefix. The Flags attribute is information for the compiler for creating a different string representation of the values—for example, setting the value 3 to a variable of DaysOfWeek results in Monday, Tuesday when you use the Flags attribute (code file EnumSample/DaysOfWeek.cs):

```
[Flags]
public enum DaysOfWeek
{
  Monday = 0x1,
  Tuesday = 0x2,
  Wednesday = 0x4,
  Thursday = 0x8,
  Friday = 0x10,
  Saturday = 0x20,
  Sunday = 0x40
}
```

With such an enum declaration, you can assign a variable multiple values using the logical OR operator (code file EnumSample/Program.cs):

```
DaysOfWeek mondayAndWednesday = DaysOfWeek.Monday | DaysOfWeek.Wednesday;
Console.WriteLine(mondayAndWednesday);
```

When you run the program, the output is a string representation of the days:

```
Monday, Wednesday
```

When you set different bits, you also can combine single bits to cover multiple values, such as Weekend with a value of 0x60. The value 0x60 is created by combining Saturday and Sunday with the logical OR operator. Workday is set to 0x1f to combine all days from Monday to Friday, and AllWeek to combine Workday and Weekend with the logical OR operator (code file EnumSample/DaysOfWeek.cs):

```
[Flags]
public enum DaysOfWeek
{
  Monday = 0x1,
  Tuesday = 0x2,
  Wednesday = 0x4,
  Thursday = 0x8,
  Friday = 0x10,
```

```
    Saturday = 0x20,
    Sunday = 0x40,
    Weekend = Saturday | Sunday,
    Workday = 0x1f,
    AllWeek = Workday | Weekend
}
```

With this in place, you can assign `DaysOfWeek.Weekend` directly to a variable, but assigning the separate values `DaysOfWeek.Saturday` and `DaysOfWeek.Sunday` combined with the logical OR operator gives the same result. The output shown is the string representation of `Weekend`:

```
DaysOfWeek weekend = DaysOfWeek.Saturday | DaysOfWeek.Sunday;
Console.WriteLine(weekend);
```

When you're working with enumerations, the class `Enum` is sometimes a big help for dynamically getting some information about enum types. `Enum` offers methods to parse strings to get the corresponding enumeration constant and to get all the names and values of an enum type.

The following code snippet uses a string to get the corresponding `Color` value using `Enum.TryParse` (code file `EnumSample/Program.cs`):

```
if (Enum.TryParse<Color>("Red", out Color red))
{
  Console.WriteLine($"successfully parsed {red}");
}
```

> **NOTE** `Enum.TryParse` *is a generic method where* `T` *is a generic parameter type. The return value is a Boolean to return* `true` *when parsing succeeded. To return the parsed enum result, the* `out` *keyword is used as a modifier of the parameter. With the* `out` *keyword, you can specify to return multiple values from a method. This keyword is discussed in the next section.*

The `Enum.GetNames` method returns a string array of all the names of the enumeration:

```
foreach (var color in Enum.GetNames(typeof(Color)))
{
  Console.WriteLine(color);
}
```

When you run the application, this is the output:

```
Red
Green
Blue
```

To get all the values of the enumeration, you can use the method `Enum.GetValues`. To get the integral value, it needs to be cast to the underlying type of the enumeration, which is done by the `foreach` statement:

```
foreach (short color in Enum.GetValues(typeof(Color)))
{
  Console.WriteLine(color);
}
```

REF, IN, AND OUT

A *value type* is passed by value; thus, the value of a variable is copied when assigned to another variable, such as when it's passed to a method. There's a way around that. If you use the `ref` keyword, a value type is passed by reference. In this section, you learn about the parameter and return type modifiers `ref`, `in`, and `out`.

ref Parameters

The following code snippet defines the method `ChangeAValueType`, where an `int` is passed by reference. Remember, the `int` is declared as struct, so this behavior is valid with custom structs as well. By default, the `int` would be passed by value. Because of the `ref` modifier, the `int` is passed by reference (using an address of the `int` variable). Within the implementation, now the variable named `x` references the same data on the stack as the variable `a` does. Changing the value of `x` also changes the value of `a`, so after the invocation, the variable `a` contains the value `2` (code file `RefInOutSample/Program.cs`):

```
int a = 1;
ChangeAValueType(ref a);
Console.WriteLine($"the value of a changed to {a}");

void ChangeAValueType(ref int x)
{
  x = 2;
}
```

Passing a value type by reference requires the `ref` keyword with the method declaration and when calling the method. This is important information for the caller; knowing the method receiving this value type can change the content.

Now you might wonder if it could be useful to pass a reference by using the `ref` keyword. Passing a reference allows the method to change the content anyway. Indeed, it can be useful, as the following code snippet demonstrates. The method `ChangingAReferenceByReference` specifies the `ref` modifier with the argument of type `SomeData`, which is a class. In the implementation, first the value of the `Value` property is changed to `2`. After this, a new instance is created, which references an object with a `Value` of `3`. If you try to remove the `ref` keyword from the method declaration, as well as the invocation of this method, after the invocation `data1.Value` has the value `2`. Without the `ref` keyword, the `data1` variable references the object on the heap and the `data` variable at the beginning of the method. After creating a new object, the `data` variable references a new object on the heap, which then contains the value `3`. With the `ref` keyword used as in the sample, the `data` variable references the `data1` variable; it's a pointer to a pointer. This way, a new instance can be created within the `ChangingAReferenceByRef` method, and the variable `data1` references this new object instead of the old one:

```
SomeData data1 = new() { Value = 1 };
ChangingAReferenceByRef(ref data1);
Console.WriteLine($"the new value of data1.Value is: {data1.Value}");

void ChangingAReferenceByRef(ref SomeData data)
{
  data.Value = 2;
  data = new SomeData { Value = 3 };
}

class SomeData
{
  public int Value { get; set; }
}
```

in Parameters

If you want to avoid the overhead of copying a value type when passing it to a method but don't want to change the value within the method, you can use the `in` modifier.

For the next sample code, the `SomeValue` struct, which contains four `int` values, is defined (code file `RefInOutSample/Program.cs`):

```
struct SomeValue
{
  public SomeValue(int value1, int value2, int value3, int value4)
  {
    Value1 = value1;
    Value2 = value2;
    Value3 = value3;
    Value4 = value4;
  }
  public int Value1 { get; set; }
  public int Value2 { get; set; }
  public int Value3 { get; set; }
  public int Value4 { get; set; }
}
```

If you declare a method where the `SomeValue` struct is passed as an argument, the four `int` values need to be copied on method invocation. When you use the `ref` keyword, you don't need a copy, and you can pass a reference. However, with the `ref` keyword, the caller might not want the called method to make any change. To guarantee that changes are not happening, you use the `in` modifier. With this modifier, a pass by reference is happening, but the compiler does not allow change to any value when the data variable is used. Data is now a read-only variable:

```
void PassValueByReferenceReadonly(in SomeValue data)
{
  // data.Value1 = 4; - you cannot change a value, it's a read-only variable!
}
```

ref return

To avoid copying the value on return of a method, you can declare the return type with the `ref` keyword and use `return ref`. The `Max` method receives two `SomeValue` structs with the parameters and returns the larger of these two. With the parameters, the values are not copied using the `ref` modifier, as shown here:

```
ref SomeValue Max(ref SomeValue x, ref SomeValue y)
{
  int sumx = x.Value1 + x.Value2 + x.Value3 + x.Value4;
  int sumy = y.Value1 + y.Value2 + y.Value3 + y.Value4;

  if (sumx > sumy)
  {
    return ref x;
  }
  else
  {
    return ref y;
  }
}
```

Within the implementation of the `Max` method, you can replace the `if/else` statement with a *conditional ref expression*. With this, the `ref` keyword needs to be used with the expression to compare `sumx` and `sumy`. Based on the result, a `ref` to `x` or to `y` is written to a `ref` local, which is then returned:

```
ref SomeValue Max(ref SomeValue x, ref SomeValue y)
{
  int sumx = x.Value1 + x.Value2 + x.Value3 + x.Value4;
  int sumy = y.Value1 + y.Value2 + y.Value3 + y.Value4;

  ref SomeValue result = ref (sumx > sumy) ? ref x : ref y;
  return ref result;
}
```

Whether the returned value should be copied or a reference should be used is a decision from the caller. In the following code snippet, with the first invocation of the `Max` method, the result is copied to the `bigger1` variable, although the method is declared to return a `ref`. There's not a compiler error with the first version (contrary to the ref parameters). You will not have any issues when the value is copied—other than the performance hit. With the second invocation, the `ref` keyword is used to invoke the method to get a `ref return`. With this invocation, the result needs to be written to a `ref` local. The third invocation writes the result into a `ref readonly` local. With the `Max` method, there's no change needed. The `readonly` used here is only to specify that the `bigger3` variable will not be changed, and the compiler complains if properties are set to change its values:

```
SomeValue one = new SomeValue(1, 2, 3, 4);
SomeValue two = new SomeValue(5, 6, 7, 8);

SomeValue bigger1 = Max(ref one, ref two);
ref SomeValue bigger2 = ref Max(ref one, ref two);
ref readonly SomeValue bigger3 = ref Max(ref one, ref two);
```

The `Max` method doesn't change any of its inputs. This allows using the `in` keyword with the parameters as shown with the `MaxReadonly` method. However, here the declaration of the return must be changed to `ref readonly`. If this change wouldn't be necessary, the caller of this method would be allowed to change one of the inputs of the `MaxReadonly` method after receiving the result:

```
ref readonly SomeValue MaxReadonly(in SomeValue x, in SomeValue y)
{
  int sumx = x.Value1 + x.Value2 + x.Value3 + x.Value4;
  int sumy = y.Value1 + y.Value2 + y.Value3 + y.Value4;

  return ref (sumx > sumy) ? ref x : ref y;
}
```

Now the caller is required to write the result to a `ref readonly` or to copy the result into a new local. With `bigger5`, `readonly` is not required because the original value received is copied:

```
ref readonly SomeValue bigger4 = ref MaxReadonly(in one, in two);
SomeValue bigger5 = MaxReadonly(in one, in two);
```

out Parameters

If a method should return multiple values, there are different options. One option is to create a custom type. Another option is to use the `ref` keyword with parameters. Using the `ref` keyword, the parameter needs to be initialized before invoking the method. With the `ref` keyword, data is passed into and returned from the method. If the method should just return data, you can use the `out` keyword.

The `int.Parse` method expects a `string` to be passed and returns an `int`—if the parsing succeeds. If the `string` cannot be parsed to an `int`, an exception is thrown. To avoid such exceptions, you can instead use the

`int`.`TryParse` method. This method returns a Boolean whether the parsing is successful or not. The result of the parse operation is returned with an `out` parameter.

This is the declaration of the `TryParse` method with the `int` type:

```
bool TryParse(string? s, out int result);
```

To invoke the `TryParse` method, an `int` is passed with the `out` modifier. Using the `out` modifier, the variable doesn't need to be declared before invoking the method and doesn't need to be initialized:

```
Console.Write("Please enter a number: ");
string? input = Console.ReadLine();
if (int.TryParse(input, out int x))
{
  Console.WriteLine();
  Console.WriteLine($"read an int: {x}");
}
```

TUPLES

With arrays, you can combine multiple objects of the same type into one object. When you're using classes, structs, and records, you can combine multiple objects into one object and add properties, methods, events, and all the different members of types. Tuples enable you to combine multiple objects of different types into one without the complexity of creating custom types.

To better understand some advantages of tuples, let's take a look at what a method can return. To return a result from a method that returns multiple results, you need to either create a custom type where you can combine the different result types or use the `ref` or `out` keywords with parameters. Using `ref` and `out` has an important restriction: you cannot use this with asynchronous methods. Creating custom types has its advantages, but in some cases, this is not needed. You have a simpler path with tuples and can return a tuple from a method. As of C# 7, tuples are integrated with the C# syntax.

Declaring and Initializing Tuples

A tuple can be declared using parentheses and initialized using a tuple literal that is created with parentheses as well. In the following code snippet, on the left side, a tuple variable `tuple1` that contains a `string`, an `int`, and a `Book` is declared. On the right side, a tuple literal is used to create a tuple with the string `magic`, the number `42`, and a `Book` object initialized using the primary constructor of the `Book` record. The tuple can be accessed using the variable `tuple1` with the members declared in the parentheses (`AString`, `Number`, and `Book` in this example; code file `TuplesSample/Program.cs`):

```
void IntroTuples()
{
  (string AString, int Number, Book Book) tuple1 =
    ("magic", 42, new Book("Professional C#", "Wrox Press"));
    Console.WriteLine($"a string: {tuple1.AString}, " +
      $"number: {tuple1.Number}, " +
      $"book: {tuple1.Book}");
  //...
}

public record Book(string Title, string Publisher);
```

When you run the application (the top-level statements invoke `IntroTuples`), the output shows the values of the tuple:

```
a string: magic, number: 42, book: Book { Title = Professional C#, Publisher =
Wrox Press }
```

> **NOTE** *There was some discussion on naming tuples using camelCase or PascalCase. Microsoft doesn't give a guideline on naming internal and private members, but with public APIs it was decided to name tuple members using PascalCase. After all, the names you specify with the tuple are public members, and these are usually PascalCase. See* `https://github.com/dotnet/runtime/issues/27939` *if you are interested in this discussion between different teams at Microsoft and the community.*

The tuple literal also can be assigned to a tuple variable without declaring its members. This way the members of the tuple are accessed using the member names of the `ValueTuple` struct: `Item1`, `Item2`, and `Item3`:

```
var tuple2 = ("magic", 42, new Book("Professional C#", "Wrox Press"));
Console.WriteLine($"a string: {tuple2.Item1}, number: {tuple2.Item2}, " +
  $"book: {tuple2.Item3}");
```

You can assign names to the tuple fields in the tuple literal by defining the name followed by a colon, which is the same syntax as with object literals:

```
var tuple3 = (AString: "magic", Number: 42,
  Book: new Book("Professional C#", "Wrox Press"));
```

With all this, names are just a convenience. You can assign one tuple to another one when the types match; the names do not matter:

```
(string S, int N, Book B) tuple4 = tuple3;
```

The name of the tuple members can also be inferred from the source. With the variable `tuple5`, the second member is a string with the title of the book. A name for this member is not assigned, but because the property has the name `Title`, `Title` is automatically taken for the tuple member name:

```
Book book = new("Professional C#", "Wrox Press");
var tuple5 = (ANumber: 42, book.Title);
Console.WriteLine(tuple5.Title);
```

Tuple Deconstruction

Tuples can be deconstructed into variables. To do this, you just need to remove the tuple variable from the previous code sample and define variable names in parentheses. The variables that contain the values of the tuple parts can then be directly accessed. In case some variables are not needed, you can use *discards*. Discards are C# placeholder variables with the name _. Discards are meant to just ignore the results, as shown with the second deconstruction in the following code snippet (code file `TuplesSample/Program.cs`):

```
void TuplesDeconstruction()
{
  var tuple1 = (AString: "magic",
    Number: 42, Book: new Book("Professional C#", "Wrox Press"));
  (string aString, int number, Book book) = tuple1;

  Console.WriteLine($"a string: {aString}, number: {number}, book: {book}");

  (_, _, var book1) = tuple1;
  Console.WriteLine(book1.Title);
}
```

Returning Tuples

Let's get into a more useful example: a method returning a tuple. The method `Divide` from the following code snippet receives two parameters and returns a tuple consisting of two `int` values. Tuple results are created by putting the methods return group within parentheses (code file `Tuples/Program.cs`):

```
static (int result, int remainder) Divide(int dividend, int divisor)
{
  int result = dividend / divisor;
  int remainder = dividend % divisor;
  return (result, remainder);
}
```

The result is deconstructed into the `result` and `remainder` variables:

```
private static void ReturningTuples()
{
  (int result, int remainder) = Divide(7, 2);
  Console.WriteLine($"7 / 2 - result: {result}, remainder: {remainder}");
}
```

> **NOTE** *When you're using tuples, you can avoid declaring method signatures with* `out` *parameters.* `out` *parameters cannot be used with* `async` *methods; this restriction does not apply with tuples.*

VALUETUPLE

When you're using the C# tuple syntax, the C# compiler creates `ValueTuple` structures behind the scenes. .NET defines seven generic `ValueTuple` structures for one to seven generic parameters and another one where the eighth parameter can be another tuple. Using a tuple literal results in an invocation of `Tuple.Create`. The tuple structure defines public fields named `Item1`, `Item2`, `Item3`, and so on to access all the items.

For the names of the elements, the compiler uses the attribute `TupleElementNames` to store the custom names of the tuple members. This information is read from the compiler to invoke the correct members.

> **NOTE** *Attributes are covered in detail in Chapter 12, "Reflection, Metadata, and Source Generators."*

DECONSTRUCTION

You've already seen deconstruction with tuples—writing tuples into simple variables. You also can do deconstruction with any custom type: deconstructing a class or struct into its parts.

For example, you can deconstruct the previously shown `Person` class into first name, last name, and age. In the sample code, the age returned from the deconstruction is ignored using discard (code file `Classes/Program.cs`):

```
//...
(var first, var last, _) = katharina;
Console.WriteLine($"{first} {last}");
```

All you need to do is create a `Deconstruct` method (also known by the name *deconstructor*) that fills the separate parts into parameters with the `out` modifier (code file `Classes/Person.cs`):

```
public class Person
{
  //...
  public void Deconstruct(out string firstName, out string lastName,
    out int age)
  {
    firstName = FirstName;
    lastName = LastName;
    age = Age;
  }
}
```

Deconstruction is implemented with the method name `Deconstruct`. This method is always of type `void` and returns the parts with multiple `out` parameters. Instead of creating a member of a class, for deconstruction you can also create an extension method as shown here:

```
public static class PersonExtensions
{
  public static void Deconstruct(this Person person, out string firstName,
    out string lastName, out int age)
  {
    firstName = person.FirstName;
    lastName = person.LastName;
    age = person.Age;
  }
}
```

> **NOTE** *With positional records, the* `Deconstruct` *method is implemented from the compiler. When you're defining a primary constructor, the compiler knows about the ordering of the parameters for the* `Deconstruct` *method and can create it automatically. With nominal records, you can create a custom implementation of the* `Deconstruct` *method similar to classes you've seen. In any case (with positional or nominal records, or with classes), you can define overloads with different parameter types as needed.*

PATTERN MATCHING

Chapter 2 covers basic functionality with pattern matching using the `is` operator and the `switch` statement. This can now be extended with some more features on pattern matching, such as using tuples and property patterns.

Pattern Matching with Tuples

The previous chapter included a sample of simple pattern matching with traffic lights. Now let's extend this sample with not just a simple flow from red to green to amber to red. . . but to change to different states after amber depending on what the previous light was. Pattern matching can be based on tuple values.

> **NOTE** *The traffic light sequences are different in many countries worldwide. With a change from amber (or yellow) to red in Canada and a few other countries, amber and red appear together to indicate a change. In most European countries, changing from red to green, the red and amber lights are displayed together for one, two, or three seconds. In Austria, China, Russia, Israel, and more, the green light starts flashing at the end of the go phase. If you are interested in the details, read* `https://en.wikipedia.org/wiki/Traffic-light_signalling_and_operation`.

The method `NextLightUsingTuples` receives enum values for the current and previous traffic light in two parameters. The two parameters are combined to a tuple with `(current, previous)` to define the switch expression based on this tuple. With the switch expression, tuple patterns are used. The first case matches when the current light has the value `Red`. The value of the previous light is ignored using a discard. The `NextLightUsingTuples` method is declared to return a tuple with `Current` and `Previous` properties. In the first match, a tuple that matches this return type is returned with `(Amber, current)` to specify the new value `Amber` for the current light. In all the cases, the previous light is set from the current light that was received. When the current light is `Amber`, now the tuple pattern results in different outcomes depending on the previous light. If the previous light was `Red`, the new light returned is `Green`, and vice versa (code file `PatternMatchingSample/Program.cs`):

```
(TrafficLight Current, TrafficLight Previous)
  NextLightUsingTuples(TrafficLight current, TrafficLight previous) =>
    (current, previous) switch
    {
      (Red, _) => (Amber, current),
      (Amber, Red) => (Green, current),
      (Green, _) => (Amber, current),
      (Amber, Green) => (Red, current),
      _ => throw new InvalidOperationException()
    };
```

With the following code snippet, the method `NextLightUsingTuples` is invoked in a `for` loop. The return value is deconstructed into `currentLight` and `previousLight` variables to write the current light information to the console and to invoke the `NextLightUsingTuples` method in the next iteration:

```
var previousLight = Red;
var currentLight = Red;
for (int i = 0; i < 10; i++)
{
  (currentLight, previousLight) = NextLightUsingTuples(currentLight,
    previousLight);
  Console.Write($"{currentLight} - ");
  await Task.Delay(1000);
}
Console.WriteLine();
```

> **NOTE** *With the statement* `await Task.Delay(1000);` *the application just waits for one second before the next statement is invoked. With top-level statements, you can directly add* `async` *methods as shown. In case you want to add this statement to a method, the method needs to have the async modifier and it is best to return a* `Task`. *This is covered in detail in Chapter 11, "Tasks and Asynchronous Programming."*

Property Pattern

Let's extend the traffic light sample again. When you're using tuples, additional values and types can be added to extend the functionality. However, at some point this doesn't help with readability, and using classes or records is helpful.

One extension to the traffic light is having different timings for the different light phases. Another extension is used in some countries: before the light changes from the green to the amber light, another phase is introduced: the green light blinks three times. To keep up with the different states, the record `TrafficLightState` is introduced (code file `PatternMatchingSample/Program.cs`):

```
public record TrafficLightState(TrafficLight CurrentLight,
  TrafficLight PreviousLight, int Milliseconds, int BlinkCount = 0);
```

The enum type `TrafficLight` is extended to include `GreenBlink` and `AmberBlink`:

```
public enum TrafficLight
{
  Red,
  Amber,
  Green,
  GreenBlink,
  AmberBlink
}
```

The new method `NextLightUsingRecords` receives a parameter of type `TrafficLightState` with the current light state and returns a `TrafficLightState` with the new state. In the implementation, a `switch` expression is used again. This time, the cases are selected using the *property pattern*. If the property `CurrentLight` of the variable `trafficLightState` has the value `AmberBlink`, a new `TrafficLightState` with the current red light is returned. When the `CurrentLight` is set to `Amber`, the `PreviousLight` property is verified as well. Depending on the `PreviousLight` value, different records are returned. Another pattern is used in this scenario—the *relational pattern* that is new with C# 9. `BlinkCount: < 3` references the `BlinkCount` property and verifies whether the value is smaller than `3`. If this is the case, the returned `TrafficLightState` is cloned from the previous state using the `with` expression, and the `BlinkCount` is incremented by `1`:

```
TrafficLightState NextLightUsingRecords(TrafficLightState trafficLightState)
  => trafficLightState switch
  {
    { CurrentLight: AmberBlink } =>
      new TrafficLightState(Red, trafficLightState.PreviousLight, 3000),
    { CurrentLight: Red } =>
      new TrafficLightState(Amber, trafficLightState.CurrentLight, 200),
    { CurrentLight: Amber, PreviousLight: Red} =>
      new TrafficLightState(Green, trafficLightState.CurrentLight, 2000),
    { CurrentLight: Green } =>
      new TrafficLightState(GreenBlink, trafficLightState.CurrentLight,
        100, 1),
    { CurrentLight: GreenBlink, BlinkCount: < 3 } =>
      trafficLightState with
        { BlinkCount = trafficLightState.BlinkCount + 1 },
    { CurrentLight: GreenBlink } =>
      new TrafficLightState(Amber, trafficLightState.CurrentLight, 200),
    { CurrentLight: Amber, PreviousLight: GreenBlink } =>
      new TrafficLightState(Red, trafficLightState.CurrentLight, 3000),
    _ => throw new InvalidOperationException()
  };
```

The method `NextLightUsingRecords` is invoked in a `for` loop similar to the sample before. Now, an instance of `TrafficLightState` is passed as an argument to the method `NextLightUsingRecords`. The new value is received from this method, and the current state is shown on the console:

```
TrafficLightState currentLightState = new(AmberBlink, AmberBlink, 2000);

for (int i = 0; i < 20; i++)
{
  currentLightState = NextLightUsingRecords(currentLightState);
  Console.WriteLine($"{currentLightState.CurrentLight},
    {currentLightState.Milliseconds}");
  await Task.Delay(currentLightState.Milliseconds);
}
```

PARTIAL TYPES

The `partial` keyword allows a type to span multiple files. Typically, a code generator of some type is generating part of a class, and having the class in multiple files can be beneficial. Let's assume you want to make some additions to the class that is automatically generated from a tool. If the tool reruns, your changes are lost. The `partial` keyword is helpful for splitting the class into two files and making your changes to the file that is not defined by the code generator.

To use the `partial` keyword, simply place `partial` before `class`, `struct`, or `interface`. In the following example, the class `SampleClass` resides in two separate source files: `SampleClassAutogenerated.cs` and `SampleClass.cs`:

```
//SampleClassAutogenerated.cs
partial class SampleClass
{
  public void MethodOne() { }
}

//SampleClass.cs
partial class SampleClass
{
  public void MethodTwo() { }
}
```

When the project that contains the two source files is compiled, a single type called `SampleClass` will be created with two methods: `MethodOne` and `MethodTwo`.

Nested partials are allowed as long as the `partial` keyword precedes the `class` keyword in the nested type. Attributes, XML comments, interfaces, generic-type parameter attributes, and members are combined when the partial types are compiled into the type. Given these two source files:

```
// SampleClassAutogenerated.cs
[CustomAttribute]
partial class SampleClass: SampleBaseClass, ISampleClass
{
  public void MethodOne() { }
}

// SampleClass.cs
[AnotherAttribute]
partial class SampleClass: IOtherSampleClass
```

```
{
  public void MethodTwo() { }
}
```

the equivalent source file would be as follows after the compile:

```
[CustomAttribute]
[AnotherAttribute]
partial class SampleClass: SampleBaseClass, ISampleClass, IOtherSampleClass
{
  public void MethodOne() { }
  public void MethodTwo() { }
}
```

> **NOTE** *Although it may be tempting to create huge classes that span multiple files and possibly have different developers working on different files but the same class, the* `partial` *keyword was not designed for this use. With such a scenario, it would be better to split the big class into several smaller classes to have a class just for one purpose.*

Partial classes can contain partial methods. This is extremely useful if generated code should invoke methods that might not exist at all. The programmer extending the partial class can decide to create a custom implementation of the partial method or do nothing. The following code snippet contains a partial class with the method `MethodOne` that invokes the method `APartialMethod`. The method `APartialMethod` is declared with the `partial` keyword; thus, it does not need any implementation. If there's not an implementation, the compiler removes the invocation of this method:

```
//SampleClassAutogenerated.cs
partial class SampleClass
{
  public void MethodOne()
  {
    APartialMethod();
  }
  public partial void APartialMethod();
}
```

An implementation of the partial method can be done within any other part of the partial class, as shown in the following code snippet. With this method in place, the compiler creates code within `MethodOne` to invoke this `APartialMethod` declared here:

```
// SampleClass.cs
partial class SampleClass: IOtherSampleClass
{
  public void APartialMethod()
  {
    // implementation of APartialMethod
  }
}
```

> **NOTE** *Prior to C# 9, partial methods had to be declared* `void`*. This is no longer necessary. However, with partial methods that do not return* `void`*, an implementation is required. This is an extremely useful enhancement, such as when using code generators. Code generators are covered in Chapter 12.*

SUMMARY

This chapter examined C# syntax for creating custom types with classes, records, structs, and tuples. You've seen how to declare static and instance fields, properties, methods, and constructors, both with curly brackets and with expression-bodied members.

In a continuation of Chapter 2, you've also seen more features with pattern matching, such as tuple, property, and relational patterns.

The next chapter extends the types with inheritance, adding interfaces, and using inheritance with classes, records, and interfaces.